Compression d'entiers par Bit Packing avec accès direct et zone de débordement

Rapport de projet de génie logiciel 2025

Di Russo Marc
M1 – Informatique

Table des matières

1 Introduction

Enoncé synthétique : ce qu'il fallait réaliser

Contexte et idée centrale. Ce projet (Java) vise à **réduire la taille mémoire et le coût de transmission** de tableaux d'entiers 32 bits tout en **conservant un accès direct** au i-ème élément (sans décompression complète), en s'appuyant sur le *bit packing* [1].

Travail demandé (3 variantes).

- **CompressionSansChevauchement** : cases de k bits *alignées* dans des mots de 32 bits ; pas de traversée de frontière de mot ;
- **CompressionAvecChevauchement** : flux binaire continu ; une valeur peut chevaucher deux mots pour maximiser l'utilisation des bits ;
- **CompressionAvecDebordement** : bit d'indicateur + contenu ; les valeurs *petites* sont codées en place, les valeurs *grandes* sont stockées dans une zone de débordement séparée.

Travail demandé (Factory). L'architecture est modulaire (`CompressionFactory` + enum `TypeCompression`) Tous les formats sont **auto-portants** (en-têtes) afin de permettre une décompression *sans état* et des tests de cohérence. Nous présentons les modèles, les choix de conception, le protocole expérimental, les résultats et leurs limites.

Travail demandé (Benchmarks). Une `BenchmarkCompression`/`MesurePerformance` pour mesurer *temps de compression/décompression*, *taille transmise* (payload et total) et *latence seuil* de rentabilité.

Exemple simple. Pour le tableau $[5, 7, 8]$, le maximum vaut 8, donc $k = 4$ bits suffisent (puisque $8 = 1000_2$). Au lieu de $3 \times 32 = 96$ bits, on peut viser $3 \times 4 = 12$ bits de *payload*.

Partie expérimentale. Chronométrer `compresser`, `decompresser` et `get(i)` ; comparer le temps gagné/perdu ; estimer à partir de quelle **latence réseau** la compression devient intéressante (seuil de rentabilité). Les résultats tiennent compte de la taille *payload* et de la taille *totale transmise* (en-tête inclus).

Exigences d'architecture et de livrable. Concevoir une architecture claire (fabrique de classes, énumération de types), un rapport et un dépôt. Dans ce projet, les en-têtes rendent les formats auto-descriptifs, et la décompression fonctionne *sans état externe*, facilitant l'évaluation et l'utilisation en transmission.

2 Modèle et rappels théoriques

Soit un tableau $X = (x_0, \ldots, x_{n-1})$ d'entiers non négatifs. On note :

$$k = \max\bigl(1, \lceil \log_2(\max X + 1) \rceil\bigr).$$

La **payload** théorique vaut $\text{totalBits} = n\,k$ et requiert $\lceil \text{totalBits}/32 \rceil$ entiers de 32 bits.

Dans l'option *débordement*, on choisit un $k' < k$ (heuristique/optimisation) et on réserve une zone *overflow* pour les valeurs $\geq 2^{k'}$. Chaque élément est encodé par un champ de largeur constante :

$$\text{largeurChamp} = 1 + \max\bigl(k', \text{bitsIndex}\bigr), \quad \text{bitsIndex} = \lceil \log_2(\text{lenOverflow}) \rceil.$$

La payload binaire devient alors :

$$\text{totalBits} = n\,\text{largeurChamp} + 32\,\text{lenOverflow}.$$

Le **seuil de rentabilité** en latence (formulation simple, au niveau des entiers 32 bits) :

$$t_{\text{break-even}} \approx \frac{T_{\text{comp}}}{N_{\text{orig}} - N_{\text{comp}}},$$

avec $N_{\text{orig}} = n$ et N_{comp} la taille transmise en entiers 32 bits (header inclus). Une version plus fine au niveau *bits* est discutée plus loin.

Bit packing et regroupement sur 32 bits [1].

- **Sans chevauchement** : on remplit des mots de 32 bits par paquets de k bits, ce qui peut introduire un *padding* si $32 \bmod k \neq 0$;
- **Avec chevauchement** : on considère un flux planaire de $n\,k$ bits, découpé ensuite en mots 32 bits ;
- **Débordement** : chaque élément porte un *bit indicateur* (0/1) et un *contenu* (valeur tronquée ou index vers la zone overflow).

Formule opérationnelle fournie par le projet. Pour exprimer un *seuil de rentabilité temporel* simple à partir des mesures agrégées, on retient par ailleurs :

$$t_{\text{rentable}} = \frac{T_{\text{compression}} - T_{\text{decompression}}}{\text{gain\_taille}},$$

où gain_taille représente la *différence de taille transmise* (par exemple en entiers 32 bits ou en octets) entre l'original et le compressé. Cette écriture est cohérente avec la logique expérimentale rapportée (voir sections 4 and 5).

3 Conception et formats de codage

3.1 Sans chevauchement

Chaque valeur occupe *exactement* k bits, alignée dans les 32 bits d'un mot ; aucun élément ne traverse deux mots. En cas de $32 \bmod k \neq 0$, un *padding* intra-mot apparaît.

Principe (bit packing sans chevauchement). On suppose que chaque entier tient sur k bits (par ex. valeurs $\in [0, 1023] \Rightarrow k = 10$). On regroupe plusieurs entiers *entiers* (non coupés) dans des mots 32 bits. Un mot peut contenir $\lfloor 32/k \rfloor$ valeurs.

Exemple. Pour $[5, 12, 31]$ et $k = 6$ (max = 63) : $5 = 000101$, $12 = 001100$, $31 = 011111$. Ici $\lfloor 32/6 \rfloor = 5$, donc **un seul int** suffit pour stocker ces trois blocs de 6 bits.

Compression.

1. Calculer k (ou `largeurBits`) à partir du max : `k = 32 - Integer.numberOfLeadingZeros(max)` ;
2. Remplir séquentiellement les mots de 32 bits en décalant (`<<`) et combinant (`|`) des blocs de k bits sans jamais les couper.

Décompression.

1. Lire chaque mot 32 bits ;
2. Extraire les blocs de k bits via un masque `mask = (1 << k) - 1`, et des décalages (`>>>`) successifs.

Accès direct get(i). Trouver le mot `indexInt = i / valeursParMot`, puis `offset = (i % valeursParMot)` ; extraire `(mot >>> offset) & mask`.

3.2 Avec chevauchement

Flux binaire continu : l'élément i commence au bit $i\,k$ et peut recouvrir 2 mots si $i\,k \bmod 32 + k > 32$.

Principe (bit packing avec chevauchement). Contrairement au cas précédent, on écrit dans un *flux binaire continu*. Une valeur peut commencer dans `int[indexInt]` et finir dans `int[indexInt+1]` si elle dépasse la frontière des 32 bits.

Compression. On maintient un compteur de bits global `bitPos`.

1. Pour chaque valeur : `indexInt = bitPos / 32`, `offset = bitPos % 32` ;
2. Insérer sur k bits par décalages (`<<`) et OU (|) ;
3. Si `offset + k > 32`, écrire la partie haute dans `int[indexInt+1]` ;
4. Incrémenter `bitPos += k`.

Décompression. Même logique : calculer `indexInt`/`offset` à partir de `bitPos` ; lire dans un ou deux mots, reconstruire, puis appliquer le masque `(1 << k) - 1`.

Accès direct get(i). Calculer `bitPos = i * k`, puis `indexInt = bitPos / 32`, `offset = bitPos % 32` ; lire les k bits requis (en combinant deux mots si nécessaire) et masquer.

3.3 Avec débordement

On code chaque élément par un bit indicateur et un contenu sur $\max(k', \text{bitsIndex})$ bits, garantissant une largeur constante, gage d'accès direct. La zone de débordement contient les valeurs hors seuil, dans l'ordre d'apparition.

Principe (gestion de débordement). Si quelques valeurs *grandes* imposent un k élevé, elles dégradent tout le tableau. L'idée est de choisir un $k' < k$ pour encoder en place la majorité des *petites* valeurs et envoyer les *grandes* dans une **zone de débordement**. Chaque élément de la zone principale est encodé sur une largeur constante

$$extlargeurChamp = 1 + \max(k',\ \text{bitsIndex}), \quad \text{où bitsIndex} = \lceil \log_2(\text{lenOverflow}) \rceil.$$

Le bit de poids fort est l'**indicateur** (0 = valeur directe sur k' bits, 1 = index vers la zone overflow sur `bitsIndex` bits).

Compression.

1. Choisir k' (heuristique) à partir de la distribution ;
2. Construire `zoneDebordement` avec toutes les valeurs $\geq 2^{k'}$ (dans leur ordre d'apparition) ;
3. Calculer `bitsIndex = ceil(log2(lenOverflow))`, puis `largeurChamp = 1 + max(k', bitsIndex)` ;
4. Pour chaque valeur :
   (a) `indicateur = 1` et `contenu = index` si valeur en overflow ; sinon `indicateur = 0` et `contenu = valeur` (tronquée sur k' bits) ;
   (b) Former `champ = (indicateur << innerWidth)` contenu|, avec `innerWidth = max(k', bitsInde` ;

(c) Écrire `champ` sur `largeurChamp` bits dans le flux (`bitPos`, `indexInt = bitPos/32`, `offset = bitPos%32`), en combinant deux mots si `offset + largeurChamp > 32`.

5. Construire la trame finale : en-tête `[MAGIC, VERSION, TYPE, tailleOriginale, largeurChamp, k', bitsIndex, lenOverflow]`, puis `zoneDebordement`, puis la *payload* packée.

Décompression. Lire l'en-tête, reconstruire le contexte et la `zoneDebordement` locale. Pour chaque élément (avec `bitPos` croissant) :

1. Extraire un `champ` sur `largeurChamp` bits (potentiellement à cheval sur deux mots) ;
2. Appliquer un masque sûr `champMask = (largeurChamp>=32)?-1:((1<<largeurChamp)-1)` et `innerMask = (innerWidth>=32)?-1:((1<<innerWidth)-1)` ;
3. `champMasked = champ & champMask`, puis `indicateur = champMasked >>> innerWidth`, `contenu = champMasked & innerMask` ;
4. Si `indicateur==1`, retourner `zoneDebordement[contenu]` ; sinon `contenu`.

Accès direct get(i)**.** Même logique que la décompression, mais en utilisant l'état interne (`donneesCompressees`, `largeurChamp`, `kPrime`, `bitsIndex`, `zoneDebordement`) pour calculer `bitPos = i * largeurChamp`, lire `champ` sur 1–2 mots, masquer et décider via l'`indicateur`.

Remarques pratiques. Utiliser des masques *sûrs* pour éviter les comportements indéfinis quand la largeur vaut 32 (par ex. conditionner `(1<<w)-1`). Vérifier les bornes de `contenu` avant d'indexer la zone de débordement. Cette variante est profitable lorsque la majorité des valeurs tiennent dans $2^{k'}$ et que la taille de la zone reste modérée.

3.4 En-tête auto-portant

Pour permettre une décompression *sans état* : `[MAGIC, VERSION, TYPE, tailleOriginale, ...]`. Pour les variantes *avec/sans chevauchement* : on ajoute k. Pour *débordement* : `largeurChamp, k', bitsIndex, lenOverflow`, puis la zone overflow, puis la payload packée.

3.5 Accès direct get(i)

— **Chevauchement** : position bit $b = i\,k$, mot $j = \lfloor b/32 \rfloor$, décalage $o = b \bmod 32$; extraire sur k bits (en combinant éventuellement deux mots).
— **Sans chevauchement** : valeursParMot $= \lfloor 32/k \rfloor$, mot $j = \lfloor i/\text{valeursParMot} \rfloor$, décalage $o = (i \bmod \text{valeursParMot})\,k$.
— **Débordement** : utiliser largeurChamp et séparer indicateur / contenu après masquage.

3.6 Choix et justification de la factory

Nous retenons une **Simple Factory** (méthode statique) `CompressionFactory.create(TypeCompression)` qui instancie l'implémentation adéquate selon un paramètre (l'énum `TypeCompression`). Ce choix est adapté car nous n'avons qu'une *famille* d'objets (des compresseurs) et un point d'entrée unique de création suffit.

Comparatif point par point.

— **Simple Factory (méthode statique)**
  — *Description* : une classe avec une méthode statique qui choisit l'instance à retourner selon un paramètre.
  — *Adaptation* : **idéal** ici : on passe `TypeCompression` (*AVEC_CHEVAUCHEMENT*, *SANS_CHEVAUCHEMENT*, *AVEC_DEBORDEMENT*) et on obtient la bonne implémentation.

  - *Intérêt* : **simple**, efficace, centralise la construction ; extension aisée via ajout d'un `enum` et d'un cas.
- **Factory Method (hiérarchie d'usines)**
  - *Description* : classe abstraite + sous-classes d'usines, une par type de produit.
  - *Pourquoi pas* : *trop lourd* ici, impliquerait autant d'usines que de compresseurs.
- **Abstract Factory**
  - *Description* : crée des *familles* d'objets corrélés (ex. : compresseur + visualiseur + mesure).
  - *Pourquoi pas* : *surdimensionnée* : une seule famille d'objets dans ce projet.
- **Builder**
  - *Description* : construction pas-à-pas pour objets riches en paramètres.
  - *Pourquoi pas* : peu de paramètres configurables dans nos compresseurs ; *non nécessaire*.

Conclusion. La *Simple Factory* offre le meilleur compromis *simplicité/cohérence*. Elle encapsule la logique d'instanciation et maintient le code client propre. Le compromis principal est un `switch` centralisé (ou mapping) à maintenir lors de l'ajout d'un nouveau type, acceptable au regard du périmètre, et conforme aux objectifs de modularité du projet.

3.7 Organisation du code

Le projet utilise une interface `Compression` avec les méthodes communes `compresser(int[])`, `decompresser(int[])` et `get(int)`. Les trois implémentations concrètes sont :

- `CompressionSansChevauchement`
- `CompressionAvecChevauchement`
- `CompressionAvecDebordement`

La sélection dynamique s'effectue via `CompressionFactory` et l'énumération `TypeCompression`. La classe `MesurePerformance` fournit utilitaires et métriques, et `Main` exécute une démonstration/benchmark.

Codage 0-x / 1-x. Dans la variante débordement, l'indicateur `0` signifie "contenu = valeur tronquée sur k' bits", l'indicateur `1` signifie "contenu = index vers la zone overflow". Des masques sûrs sont employés pour éviter tout dépassement de 32 bits.

4 Mesures et protocole

Protocole. Pour chaque méthode : W itérations de *warmup* (JIT), puis R itérations mesurées avec `System.nanoTime()`. À chaque itération, vérifier `compresser` $\Rightarrow$ `decompresser` = original. Les résultats sont agrégés (moyenne, optionnellement écart-type) et affichés par `BenchmarkCompression/Main`.

Métriques. Temps moyens de compression/décompression (en ns et affichage en μs), tailles transmises (en entiers 32 bits et en bits), ratios **payload-only** et **total** (header inclus), et calcul du seuil t de rentabilité (voir section 2). Des messages explicatifs signalent les cas non rentables (notamment pour *SansChevauchement*).

Environnement. JDK. Chronométrage Java standard.

Visualisation. Tableaux alignés sur les jeux décrits en section C. Pour chaque catégorie, on présente :

- **Vue taille** : *payload-only* (ints), coût d'en-tête (%), taille totale (ints) ;
- **Vue temps** : temps moyens compresser/décompresser (ns) ;
- **Seuil de rentabilité** : t en ns/int selon la définition de section 2.

Les jeux couverts correspondent exactement à l'annexe (section C) :

- Aléatoire ($n = 8$, min=0, max=1023) ;
- Aléatoire ($n = 8$, min=0, max=1 000 000) ;
- Aléatoire ($n = 32$, min=0, max=65 535) ;
- Synthétique ($n = 256$, valeurs $\in [0..15]$) ;
- Synthétique ($n = 1024$, valeurs $\in [0..3]$) ;
- Fixes : $[5, 12, 31, 7, 15, 1023]$ et $[5, 12, 31, 7, 15, 1023, 2000, 999999]$.

5 Résultats et analyses

Nous présentons des résultats pour (i) petits jeux non rentables (header dominant), (ii) cas amortis ($n = 256$, $k \approx 4$), (iii) cas très rentables ($n = 1024$, $k \approx 2$).

Lecture recommandée. Toujours distinguer **payload-only** (ce qui reflète le bit packing) et **total transmis** (header inclus), seul pertinent pour la transmission.

Observations clés.

- Le header devient négligeable pour n grand ; k petit favorise fortement le gain.
- Sans chevauchement, le padding si $32 \bmod k \neq 0$ dégrade *légèrement* par rapport au flux continu.
- Débordement n'aide que si beaucoup de valeurs tiennent dans $2^{k'}$ et si la zone reste modérée.
- Le seuil t baisse fortement quand le ratio compressé/original diminue.

Constats spécifiques au projet.

- **Chevauchement** et **Débordement** se sont avérés efficaces sur les jeux *à petits k* ou avec forte proportion de valeurs *petites* (ratio observé jusqu'à $\approx 0{,}625$ selon les cas).
- **SansChevauchement** n'est généralement pas rentable en présence de grands entiers (padding et largeur k élevée), ce que le benchmark explicite.
- Les résultats sont **cohérents avec la théorie** : lorsque n augmente ou lorsque k diminue, la part du header dans le total devient marginale et le gain augmente.

6 Discussions et limites

Hypothèses. Entiers non négatifs ; formats 32 bits ; accès direct prioritaire ; en-tête autoportant.

Limites. Heuristique pour k' ; index d'overflow basé sur première occurrence (peut être amélioré) ; pas d'optimalité garantie ; cas non rentables pour petits tableaux ou débordement massif ; dépendance à la distribution des valeurs.

Validité des mesures. JIT/GC/thermique/OS ; répéter et lisser. Mentionner les jeux de données et les seeds.

Sécurité/robustesse. Vérifier les bornes sur `get(i)` ; tolérance aux fichiers corrompus (MAGIC/VERSION/TYPES), masques sûrs.

Difficultés rencontrées et correctifs.

- **Débordement** : correction d'un défaut lors de la décompression (extraction *après* masquage du champ à `largeurChamp`) qui pouvait conduire à des erreurs (par ex. exceptions ou incohérences d'intégrité) ;
- **Structures** : remplacement de collections `ArrayList` par des `int[]` pour éviter surcoûts et simplifier l'interface binaire ;
- **Affichage** : correction de l'affichage de tableaux (éviter la forme `[I@xxxx`) pour une lecture claire du contenu ;
- **API** : ajout de getters/setters cohérents dans les classes concernées ;
- **Pédagogie** : messages explicatifs quand la compression est non pertinente (notamment pour *SansChevauchement*).

7 Conclusion

Nous avons conçu, implémenté et évalué trois variantes de bit packing avec accès direct, y compris une gestion de débordement et un format auto-portant pour la transmission. Les mesures montrent des gains substantiels dès que n est modérément grand et k petit, et précisent un seuil de rentabilité réaliste. En synthèse :

- **CompressionAvecChevauchement** et **CompressionAvecDebordement** sont efficaces dans les distributions favorables (petits k ou majorité de petites valeurs) ;
- **CompressionSansChevauchement** est surtout pertinente pour des petits entiers et des alignements favorables ;
- Le protocole de benchmark est stable (warmup, répétitions) et met en évidence la distinction *payload vs total* ;
- L'architecture est modulaire et extensible via la factory et l'énumération de types.

A Spécifications détaillées (formats et masquages)

A.1 En-têtes

Commun `[MAGIC (0x42505431), VERSION, TYPE, tailleOriginale, ...]`

Sans/avec chevauchement + `k`

Débordement + `largeurChamp, k', bitsIndex, lenOverflow, zoneOverflow[lenOverflow]`

A.2 Masquage et décalage

- Masque k : mask $= (k \geq 32)? -1 : (1 \ll k) - 1$;
- Fusion bits entre deux mots si dépassement ;
- Débordement : champ masqué à `largeurChamp`, puis `indicateur = champ >> innerWidth`, `contenu = champ & ((1<<innerWidth)-1)`.

A.3 Types et décompression sans état

Le champ `TYPE` correspond à l'énumération `TypeCompression` du code Java. Tous les décompresseurs lisent d'abord l'en-tête pour reconstruire le contexte (extttk, `largeurChamp`, `k'`, `bitsIndex`, etc.) puis opèrent **sans dépendre d'un état externe**.

A.4 Codage indicateur 0/1

— `0-x` : la valeur est *contenue* sur k' bits dans le champ ;
— `1-x` : *index* vers la zone de débordement (`zoneOverflow`).

B Extraits de code

B.1 Accès direct (chevauchement)

```
int bitPos = i * k;
int indexInt = bitPos / 32;
int offset = bitPos % 32;
int val = (comp[indexInt] >>> offset);
if (offset + k > 32) val |= (comp[indexInt+1] << (32 - offset));
val &= (k >= 32) ? -1 : (1 << k) - 1;
```

B.2 Champ débordement (extrait exact du projet)

```
// Extrait de CompressionAvecDebordement.decompresser(...)
int champ = (compresse[dataStart + indexInt] >>> offset);
if (offset + largeurChampLocal > 32 && dataStart + indexInt + 1 < compresse
    .length) {
  champ |= (compresse[dataStart + indexInt + 1] << (32 - offset));
}

int innerWidth = Math.max(kPrimeLocal, bitsIndexLocal);
int champMask = (largeurChampLocal >= 32) ? -1 : ((1 << largeurChampLocal)
    - 1);
int innerMask = (innerWidth >= 32) ? -1 : ((1 << innerWidth) - 1);

int champMasked = champ & champMask;
int indicateur = champMasked >>> innerWidth;
int contenu = champMasked & innerMask;
```

B.3 Accès direct en débordement (get)

```
// Extrait de CompressionAvecDebordement.get(int i)
int bitPos = i * largeurChamp;
int indexInt = bitPos / 32;
int offset = bitPos % 32;

int champ = (donneesCompressees[indexInt] >>> offset);
if (offset + largeurChamp > 32 && indexInt + 1 < donneesCompressees.length)
    {
  champ |= (donneesCompressees[indexInt + 1] << (32 - offset));
}

int innerWidth = Math.max(kPrime, bitsIndex);
int champMask = (largeurChamp >= 32) ? -1 : ((1 << largeurChamp) - 1);
int innerMask = (innerWidth >= 32) ? -1 : ((1 << innerWidth) - 1);

int champMasked = champ & champMask;
int indicateur = champMasked >>> innerWidth;
int contenu = champMasked & innerMask;
```

B.4 Factory (sélection dynamique)

```
// Code client
Compression c = CompressionFactory.create(TypeCompression.
    AVEC_CHEVAUCHEMENT);
int[] compresse = c.compresser(original);
int[] decompresse = c.decompresser(compresse);
int v = c.get(42);
```

```
// Extrait de CompressionFactory
public static Compression create(TypeCompression type) {
  switch (type) {
    case AVEC_CHEVAUCHEMENT:
      return new CompressionAvecChevauchement();
    case SANS_CHEVAUCHEMENT:
      return new CompressionSansChevauchement();
    case AVEC_DEBORDEMENT:
      return new CompressionAvecDebordement();
    default:
      throw new IllegalArgumentException("Type de compression inconnu: " +
          type);
  }
}
```

C Données de benchmark (extraits)

Jeux de données testés.

- **Aléatoire (n=8, min=0, max=1023)** : petits nombres, compression potentielle forte ;
- **Aléatoire (n=8, min=0, max=1 000 000)** : grande amplitude, compression difficile ;
- **Aléatoire (n=32, min=0, max=65 535)** : taille moyenne et valeurs 16 bits ;
- **Synthétique (n=256, valeurs** $\in [0..15]$**)** : $k \leq 4$ bits, en-tête amorti, cas rentable attendu ;
- **Synthétique (n=1024, valeurs** $\in [0..3]$**)** : $k = 2$ bits, cas très rentable ;
- **Fixe** : $[5, 12, 31, 7, 15, 1023, 2000, 999999]$ (largeurs hétérogènes) ;
- **Fixe (sous-ensemble)** : $[5, 12, 31, 7, 15, 1023]$.

Jeu	n	k	ints (total)	ints (payload)
Aléatoire	8	10	8	3
[0..15]	256	4	37	32
[0..3]	1024	2	69	64

Remarques : affichage en ns et μs ; ratio observé jusqu'à $\approx 0{,}625$ sur certains jeux favorables. Voir le programme pour les figures complètes et la méthodologie de mesure.

Récapitulatif des paramètres des jeux (sans résultats numériques)

oprule Jeu	n	min	max	Type	Objectif
Aléatoire	8	0	1023	test	compression potentielle forte
Aléatoire	8	0	1 000 000	test	amplitude élevée, peu rentable
Aléatoire	32	0	65 535	test	valeurs 16 bits
[0..15]	256	0	15	synthétique	amortir l'en-tête ($k \leq 4$)
[0..3]	1024	0	3	synthétique	cas très rentable ($k = 2$)
Fixe	8	–		exemple	valeurs hétérogènes

Références

[1] *Bit packing of integers*. https://en.wikipedia.org/wiki/Bit_pack.